Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States

Anonymous Author(s)
Submission Id: XX

Abstract

Deployed distributed systems frequently encounter faults that trigger bugs. Testing a system against various faults offline is crucial. While fault injection testing becomes a popular practice, existing efforts focus on simple injection of coarse-grained faults such as node crashes. They fail to expose subtle bugs, especially the notorious partial failures.

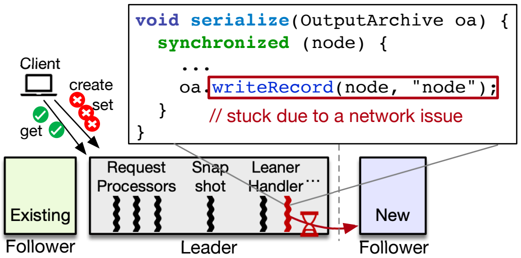
This paper presents LEGOLAS, a distributed system fault injection framework aiming to expose complex fault-triggered bugs. LEGOLAS addresses the challenges of how to precisely inject fine-grained faults, and how to efficiently explore the large number of fault injection choices. LEGOLAS first uses program analysis to instrument injection hooks tailored to a system. It further introduces a novel notion of *abstract states* and *automatically* infers abstract states from concrete code. During testing, LEGOLAS leverages the inferred abstract states to make careful injection decisions and efficiently expose bugs. We applied LEGOLAS on the latest releases of five popular, extensively tested distributed systems. LEGOLAS found 16 new bugs that result in partial failures.

1 Introduction

Deployed distributed systems frequently encounter faults in the underlying hardware as well as dependent software services. While these systems contain much fault handling code, an unexpected fault can still expose bugs. Indeed, real-world distributed system outages are often triggered by some fault events [12, 17, 19, 34]. Finding fault-triggered bugs early in testing is desirable to prevent later production damages.

Fault injection testing thus becomes a popular practice with active research [3, 9, 10, 16, 18, 26, 27, 30, 38]. To name a few, the OpenStack fault injector [27] instruments the communication libraries to inject crash or network partition; PREFAIL [26] allows testers to specify policies for injecting multiple hardware faults; CrashTuner [38] injects node crashes during meta-info variable accesses.

Despite these fruitful efforts, many complex fault-triggered bugs still escape testing, especially those leading to partial failures [14, 15, 24, 25, 36], which are notorious in production distributed systems. Figure 1 shows a real example from a ZooKeeper deployment. The clients experienced many timeouts in `create` and `set` requests, but requests like `get` could still succeed. The leader was also alive during this failure. The root cause is that a partial network fault occurred while the leader was serializing a snapshot in a thread to a new follower, but this operation is performed inside a synchronized block. As another example, users reported [5–7] that in their Kafka deployments, a broker could no longer return to *In-Sync Replica* status. This failure was caused by the improper handling of an exception in invoking an API of the ZooKeeper service that Kafka depends on.



```
void serialize(OutputArchive oa) {
  synchronized (node) {
    ...
    oa.writeRecord(node, "node");
  }     // stuck due to a network issue
}
```

Figure 1. A real ZooKeeper production incident [44] triggered by a partial network fault, which caused the `writeRecord` operation to be stuck while holding a lock. Thus, threads that process update requests are blocked. Developers fixed the bug by making a copy of the `node` object, and serializing the copy without the lock.

Exposing failures like the above examples in testing poses two key challenges. First, they only occur under subtle fault conditions, such as a partial fault that affects some but not all operations [8, 36], transient slowness [20, 23], or microburst [28]. It is also common for the fault conditions to arise from a dependent component, *e.g.*, a user-defined exception in an RPC due to errors in a remote component. Current solutions focus on simple, system-agnostic faults such as node crashes and complete network disconnection.

Second, these failures require careful choices of when and where the fault occurs. Modern distributed systems have a large number of fault injection points, but they are also designed to be robust. Thus, they can tolerate an injected fault (*e.g.*, with retries) or only experience an *expected* failure (*e.g.*, abort after a fault in reading a critical file). To expose the aforementioned ZooKeeper failure, a transient network latency increase must be injected while a new follower is requesting a snapshot from the leader. Injecting the fault at other times or other locations are ineffective. A random injection choice, which is commonly used in existing solutions, will have a high chance of missing the buggy point.

This paper presents LEGOLAS, a fault injection framework for large distributed systems to close the current gaps and efficiently expose partial failure bugs. LEGOLAS first uses program analysis to enable fine-grained and system-specific fault injection. It statically identifies the fault conditions at the code instruction level and instruments hooks to precisely control the injection of subtle faults *within* a system.

With more faults to consider, the challenge of large fault injection space becomes even more pronounced, calling for advanced injection decision algorithms. While conceptually similar, this challenge is orthogonal to the state space explosion problem in the distributed systems model checking literature [21, 31, 40, 49]. The latter concerns efficiently testing the combination of *concurrent events*. The common technique there is to leverage symmetry properties to avoid testing all combinations of multiple events. In our case, there are enormous individual injection points even without concurrency effect. Thus, such reduction techniques cannot help.

To address this challenge, our insight is that production bugs occur in unusual conditions—if a bug occurs in common scenarios, existing testing likely has exposed it. By checking if the system reaches an unusual condition, we can *selectively* inject faults. Unfortunately, we do not know beforehand whether a program point is unusual or not. Relying on heuristics, such as only injecting faults when the program is inside a critical section, can be *ad-hoc* and miss many failure-inducing conditions. We should still systematically explore the injection choices to achieve generality and completeness.

Based on this insight, LEGOLAS introduces a novel notion of *abstract state* to guide systematic but efficient exploration of the fault injection space. The basic idea is to use system states to *group* injection points. A system's state can be represented by its variables and concrete values. This representation, however, is massive for large systems, and it would make almost all injection points appear in unique groups. For fault injection, we need a more high-level state representation, in which multiple injections likely yield similar effect.

LEGOLAS uses a simple yet novel static analysis that *automatically infers* abstract states from the target system code. The automation is feasible because developers usually leave clear hints in the code about abstract states—the system checks one or more state variables' concrete values in a branch to see if an important condition occurs; if so, it performs some significantly different action. An abstract state thus can indicate that a system enters a unique stage of service, *e.g.*, request parsing, snapshotting, and leader election.

Specifically, LEGOLAS first infers concrete state variables in a system. It then identifies code blocks that are control-dependent on some concrete state variable. It finally creates a mnemonic abstract state variable for each such block, which will be set when the program execution reaches that point.

Leveraging the inferred abstract states, LEGOLAS can efficiently explore the injection space. During testing, the injection hooks LEGOLAS instruments dispatch queries to the LEGOLAS controller. The controller checks the system's current abstract states and decides whether to grant an injection or not. Essentially, LEGOLAS enables *stateful* fault injection.

We design a stateful injection decision algorithm called *budgeted-state-round-robin* (bsrr). Other stateful policies are also feasible, and it is easy to add and switch policies in LEGOLAS. Compared to the straightforward *new-state-only* policy, *bsrr* is robust to tolerate potential inaccuracies in the abstract state analysis. It also reduces biases in injections.

We have built an end-to-end prototype for the LEGOLAS framework, including the static analyzer, fault injection controller, workload driver, and failure checkers.

We apply LEGOLAS to five large distributed systems: Kafka, ZooKeeper, HDFS, HBase and Cassandra. LEGOLAS automatically instruments these systems and extracts abstract states without special tuning. We run fault injection experiments on these systems' latest releases, which are rather mature and have been extensively tested. Still, LEGOLAS finds 16 new partial failure bugs using the *bsrr* algorithm, with a median time of 58 minutes. We have reported the bugs to developers. Four reports are marked as critical, ten reports are marked as major, and two are marked as normal. Ten reports have been explicitly confirmed by developers. We also compare LEGOLAS (*bsrr*) with state-of-the-art solutions and other policies. The best of them is the *new-state-only* policy, which also leverages our inferred abstract states and exposes six bugs. Other baselines perform worse, *e.g.*, random injection only exposes three bugs with a median of 362 minutes.

In summary, this paper makes the following contributions:

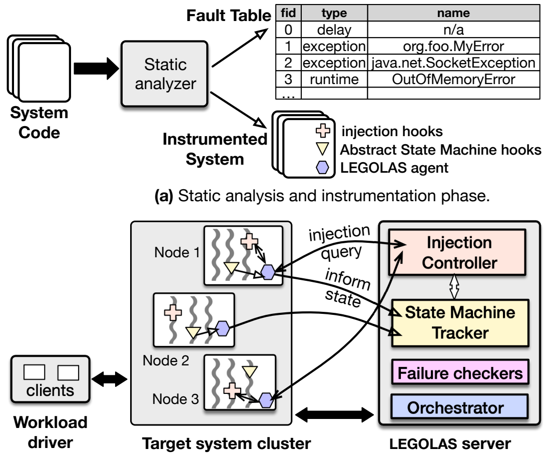
- We propose a new approach that uses program analysis to enable in-situ injection of fine-grained and system-specific faults for exposing partial failure bugs.
- We introduce a novel concept of abstract state and a method that automatically infers abstract states from a given system's code. We further design a new decision algorithm that effectively leverages the inferred abstracts to guide efficient fault injections.
- We build an end-to-end fault injection framework LEGOLAS. We evaluate LEGOLAS on large distributed systems and find 16 new partial failure bugs.

The source code of LEGOLAS will be available at https://<url> (redacted for review anonymity).

2 Background and Motivation

In performing fault injection, current solutions tend to treat the target system as a black-box node. They simulate system-agnostic faults such as node crashes *externally* at the library or execution environment, *e.g.*, by using process kill, iptable manipulation, or using a special file system [2, 16]. While this external injection approach is suitable for the goals of exposing distributed protocol bugs or crash-recovery bugs, it imposes significant limitations on the complexity of faults being tested. It is infeasible or difficult to inject partial faults or program-specific errors common in production environment.

Algorithms that decide whether and when to inject what fault are also crucial for fault injection effectiveness. Current solutions often use simple policies, such as random or exhaustive injections, or rely on manual specifications from



fid	type	name
0	delay	n/a
1	exception	org.foo.MyError
2	exception	java.net.SocketException
3	runtime	OutOfMemoryError
…		

(a) Static analysis and instrumentation phase.

(b) Dynamic fault injection testing phase.

Figure 2. Fault injection workflow with Legolas.

users. A few solutions [13, 38] use heuristics to inject faults in special code regions that are assumed to be likely buggy, such as when accessing meta-info variables. These heuristics are restricted to only specific programming patterns and a particular bug type such as crash-recovery bugs.

3 Overview of Legolas

Legolas is an end-to-end fault injection testing framework for large distributed systems. It aims to catch a wide range of bugs missed by existing solutions, such as the motivating example in Figure 1, which require complex faulty conditions to trigger and cause failure symptoms beyond crashes.

Approach. Unlike current solutions that take a system-agnostic approach, Legolas uses program analysis techniques to achieve *implementation-aware* fault injection. By analyzing the target system code, Legolas can identify the system-specific faulty conditions and enable injection at a fine granularity. Moreover, it can gather information from the code to make careful decisions for efficient exploration of the injection space.

A design principle we follow is to make Legolas generally applicable to a new system without much tuning effort. The programming paradigms, bugs, and faults are diverse across systems. If a solution requires too much domain knowledge or *ad-hoc* heuristics, not only will it be fragile when applied to a new system, but also will it miss many potential bugs.

Therefore, Legolas uses program analysis to extract only basic information that is obtainable across different systems without difficulties. At runtime, Legolas systematically explores the injection space. It leverages the extracted information to make *adaptive* decisions.

Workflow. Figure 2 shows the workflow with Legolas. Legolas performs *in-situ* injections. For a given system, its analyzer parses all the statements in the code. It identifies the potential faulty conditions for each program point (Section 4.1). If a program point may encounter a fault, Legolas instruments an injection hook (Section 4.2). A thin *agent* will manage the injection hooks and query a controller to apply faults.

In addition to instrumenting injection hooks, Legolas's static analysis phase also extracts abstract states for each system module (Section 5.3). It inserts mnemonic abstract state variables in the corresponding code locations, which will be set at runtime if reached.

An orchestrator manages the dynamic fault injection experiment, which consists of a series of trials. In each trial, the orchestrator starts the instrumented system and run the workload driver to exercise the system. When a process reaches an injection hook, the Legolas agent queries the controller, which decides whether to grant the injection or not.

The Legolas server also maintains abstract state machines for each process. When a process enters a new abstract state, the Legolas agent informs a state machine tracker. The controller leverages the state information in making decisions (Section 5.4). Legolas runs the failure checkers (Section 5.6) to determine the injection outcome.

4 Extract and Instrument Injection Points

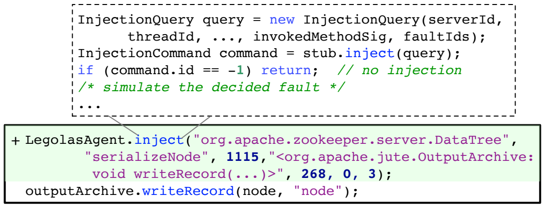
This section describes how Legolas addresses two questions: 1. *what faults to inject?* 2. *how to inject a fault?*

4.1 Extract Faulty Conditions

The faults different systems may encounter vary. For comprehensive fault injections, Legolas statically analyzes a given system's code. It locates each call instruction and examines its invoked method to extract all potential fault conditions, including standard library exceptions and custom error types. To do so, it can leverage the method signature, which typically includes the exceptions the method may throw. However, two challenges need to be addressed.

First, a method may internally throw some exceptions that are not declared in the signature. Languages such as C++ also do not enforce or support exception specification in method signature. To handle this situation, Legolas inspects the method body and deduces the exceptions. However, it cannot simply collect the exceptions in the throw instructions. This is because the method may have an exception handler that catches the exception. Legolas analyzes the error handlers in the function to determine if an exception may be (i) caught and handled; (ii) caught and re-thrown; (iii) uncaught. Only (ii) and (iii) are treated as the true method-level exceptions.

Second, due to polymorphism and interface, some call site of a method may be *impossible* to encounter an exception declared in the method signature. This is especially problematic with I/O related exceptions. Consider `void dump(OutputStream out) throws IOException`, which is declared this way because argument `OutputStream` is an abstract class with `IOException` in its methods' signatures. However, if a call site of `dump` passes a `ByteArrayOutputStream` object as an argument, injecting an `IOException` would create an invalid scenario. Legolas designs an inter-procedural, context-sensitive analysis to check call instructions with potential `IOException`. It deduces whether the objects (argument, return, field, class) associated



```
InjectionQuery query = new InjectionQuery(serverId,
      threadId, ..., invokedMethodSig, faultIds);
InjectionCommand command = stub.inject(query);
if (command.id == -1) return;  // no injection
/* simulate the decided fault */
...
```

```
+ LegolasAgent.inject("org.apache.zookeeper.server.DataTree",
       "serializeNode", 1115,"<org.apache.jute.OutputArchive:
        void writeRecord(...)>", 268, 0, 3);
  outputArchive.writeRecord(node, "node");
```

Figure 3. Injection hook added for the example in Figure 1.

with a call site may come from definition points with known in-memory object types, and ignores the fault if so.

Besides exceptions, the fault condition could also be a delay. All operations could in theory experience some delay. In practice, mild delays are benign and developers need evidence to explain the delay. LEGOLAS by default only considers function calls that involve I/O as delay injection candidates.

The final output of the analysis is a metadata called *fault table*. Each unique fault is assigned an id, type, source, and name. The fault table is referenced by the injection hooks as well as the fault injection controller in the LEGOLAS server.

4.2 Instrument Injection Hooks

Unlike current solutions that simulate faults externally in a library or special execution environment, LEGOLAS performs injection within the target system with instrumented hooks.

This *in-situ* approach eases fault simulation without requiring a special environment. It also gives precise control to simulate subtle faults, *e.g.*, partial disk failure that only affects some file operations; only some RPC is blocked due to an overloaded server. In addition, this approach directly supports simulating custom errors. While many program errors may be caused by some environment fault, it can be difficult to simulate them if we only inject faults externally. For example, a method in the program may throw an exception only when all three consecutive retries of a specific connection fail.

For a program operation to be injected, LEGOLAS instruments an injection hook. LEGOLAS also emits a thin *injection agent* to package into the target system. At runtime, when an injection hook is reached, the agent creates a query to a controller (Figure 3), which includes the interposed operation, associated fault ids, server id, name and id of the current thread. If the injection is granted, the injection agent looks up the fault id in the fault table. If the fault type is exception, the injection agent constructs an instance based on the exception name, and throws it before the injection hook returns.

Automatically creating an exception instance to throw is a non-trivial task. Some custom exception type includes complex arguments and compositions. LEGOLAS analyzes the constructor and recursively reduces complex arguments to primitive types. It then creates an exception instance by assigning the primitive fields with default values. For delay, the agent simply invokes the thread sleep function.

Where to add the injection hooks also requires careful considerations. The straightforward way is to instrument each call instruction that LEGOLAS analyzes to possibly encounter a fault (§4.1). Suppose `foo()` is analyzed to possibly throw `MyError`, but that is only because `foo()` internally calls `bar()`, which can throw that error. If we inject `MyError` at all calls to `foo()`, we later need to explain why and when `foo()` will encounter this error. For deep call chains, such injections can make the reasoning difficult and may turn out to be invalid.

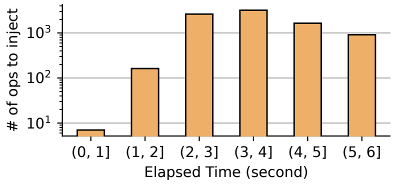


Figure 4. `IOException` injection candidates in running ZooKeeper.

To address this issue, LEGOLAS instruments as deep as possible. It identifies faults that *originate* from a method through explicit `throw` statement. It then only instruments calls to either a method that has a non-empty list of such faults, or an external function. If the called method is from an interface or abstract class, LEGOLAS injects at the caller's call sites to handle potential invalid injections (§4.1).

5 Abstract State Guided Fault Injection

This section describes how LEGOLAS addresses the question of *how to make fault injection decisions?*

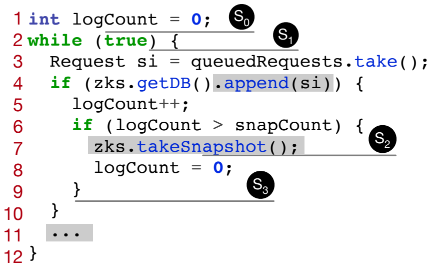
A key challenge in fault injection for distributed systems is the enormous injection choices. Figure 4 shows the dynamic injection points for only `IOExcetpion` fault type in several seconds of ZooKeeper's execution. Operations that are candidates for injection reach hundreds to thousands per second.

Moreover, modern distributed systems are designed with extensive resilience mechanisms, such as replication, retries, and cache. Thus, most fault injections would not expose unexpected bugs. For example, in ZooKeeper, a leader process manages a thread for each follower; if a network fault occurs to one follower, the corresponding thread exits, but the leader re-creates a new thread once the error is gone and the follower re-connects. If an error occurs when a server sends a response to a client, the handler just logs the error and continues. The client will find out the response loss and react properly.

5.1 Grouping Injections Based on System State

To tackle this challenge, we observe that a lot of the injection attempts are redundant in that they repeatedly test the same or similar scenarios. The redundancies arise for various reasons. For example, one injection point is invoked in a loop; multiple injection points share the same error handling logic, have the same dependencies, or all reside in an extensively tested class.

Take a ZooKeeper code snippet in Figure 5 as an example. Suppose we are injecting faults on I/O operations. There are numerous injection points here, including program points inside the called functions. Line 4 (the `append` operation) gets executed at each loop iteration, while line 7 (the `takeSnapshot` operation) only occurs occasionally. If the injection is agnostic to this code, we could make injection decisions that keep granting injection at line 4, and miss injection inside line 7.



```
 1 int logCount = 0;
 2 while (true) {
 3   Request si = queuedRequests.take();
 4   if (zks.getDB().append(si)) {
 5     logCount++;
 6     if (logCount > snapshotCount) {
 7       zks.takeSnapshot();
 8       logCount = 0;
 9     }
10   }
11   ...
12 }
```

Figure 5. The bug in Figure 1 occurs inside a call chain from line 7. The grayed areas are code regions containing I/O operations.

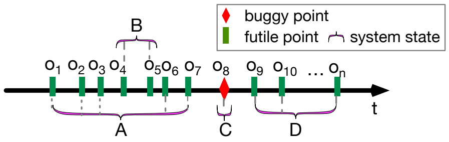

buggy point
futile point
system state
B
o_1 o_2 o_3 o_4 o_5 o_6 o_7 o_8 o_9 o_{10} … o_n
t
A
C
D

Figure 6. Group the injection points by the state they appear in.

If we instead inspect the system code, we can avoid redundant injections. For the previous example, we could realize that the system enters a rare state (snapshotting) when it reaches line 7. That injection point could be of high interest. However, we should not just look for rare states, because defining rareness is tricky. Heuristics like frequency threshold are fragile. Moreover, an injection point in a rare state is not necessarily buggy. Similarly, an injection point in a common state is not necessarily innocent. For example, a bug may be exposed with a fault occurring when the `append` call in the common state is executed for the fourth time.

Our high-level idea to tackle the large injection space challenge is to *group* the injection points based on the underlying system state (Figure 6). Grouping helps avoid being indiscriminate when making injection decisions. The injection points that lie in the same group of state are hypothesized to yield similar outcomes if injected, while the injection points in different groups may yield different outcomes. But we still explore the injection space *systematically*. That is, if there were four chances, we try to explore injections in all four states, instead of spending the budget only in one state.

5.2 State Representation Choices

The next question is *how to define the system state for effective grouping?* Unlike distributed protocols that have specifications, determining the state representation for complex system implementation is not easy. The complete execution states—the program counter, stack traces, and memory snapshot—are obviously too excessive. A more reasonable representation is to use some key state variables (SV). A value change of these variables then could indicate the system is in a different state. This representation, however, can still be excessive.

Take Figure 5 as an example. If we treat the `logCount` as a state variable (SV), the value is incremented for n times, and each increment is counted as a new state. Using such a representation not only requires frequent state tracking, but also degrades the injection point grouping to be useless.

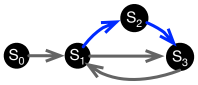


Figure 7. State machine with abstract states for Figure 5. Each state is an abstraction over the concrete state variable `logCount`.

Algorithm 1: Infer abstract state variables

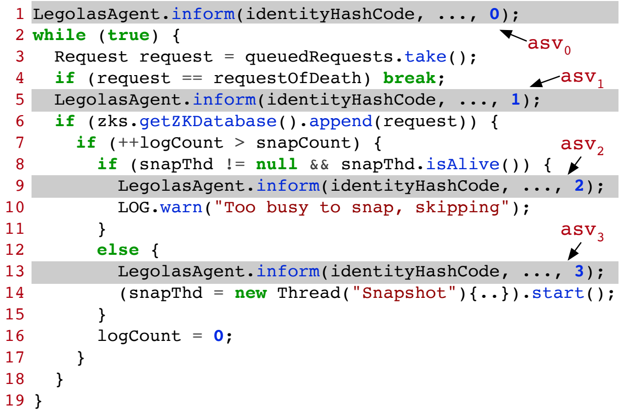
```
Function InferASV(task_class):
    csv_list ← InferCSV(task_class);
    task_method ← getTaskMethod(task_class);
    dep_graph ← buildDependence(task_method, csv_list);
    asv_locations ← empty;
    Process(task_method.body(), dep_graph, false);
Function Process(instructions, dep_graph, flag):
    inst ← instructions.begin();
    hasAction ← false;
    while inst ≠ instructions.end() do
        if isBranch(inst) then
            <cond, blocks, next> ← parseBranch(inst);
            if dep_graph.contains(cond) then
                for block ← blocks do
                    Process(block.body(), dep_graph, true);
                end
            end
            inst ← next;
        else
            hasAction ← hasAction | isAction(inst);
            inst ← inst.next();
        end
    end
    if hasAction and flag then
        asv_locations.add(instructions.begin());
```

The key reason is that some values in a concrete state variable do not imply a significant change, at least for fault injection purposes. All the increment-by-one value transitions of `logCount` while `n<=snapCount` indicate the same information about the system, while only the transition of `n>snapCount` indicates something new (starting to snapshot).

Essentially we need a more high-level representation than concrete state variables (SV), which we define as **abstract state variable** (ASV). The intuition behind ASVs is that they represent some stages of service in a system. For the example in Figure 5, a natural way to define the abstract state is to divide the execution into four stages—S_0 to S_3. Figure 7 shows the corresponding abstract state machine. This simpler representation can recognize when the system starts to do snapshot (state S_2). In turn, they can effectively group the injection points to make fault injection efficient.

5.3 Infer Abstract State Variables

The LEGOLAS analyzer uses a simple yet novel method to automatically infer ASVs in a system. The feasibility of the



```java
LegolasAgent.inform(identityHashCode, ..., 0);
while (true) {
  Request request = queuedRequests.take();
  if (request == requestOfDeath) break;
  LegolasAgent.inform(identityHashCode, ..., 1);
  if (zks.getZKDatabase().append(request)) {
    if (++logCount > snapCount) {
      if (snapThd != null && snapThd.isAlive()) {
        LegolasAgent.inform(identityHashCode, ..., 2);
        LOG.warn("Too busy to snap, skipping");
      }
      else {
        LegolasAgent.inform(identityHashCode, ..., 3);
        (snapThd = new Thread("Snapshot"){..}).start();
      }
      logCount = 0;
    }
  }
}
```

Figure 8. The ASVs LEGOLAS infers for Figure 5.

automation is based on our insight that developers usually already encode sufficient hints about ASVs. In particular, developers checks one or more state variables (SV) in a branch, and if certain condition occurs, the system performs some action, *i.e.*, `if (func(state_var1, var2, ...)) { do_action1(); }`.

LEGOLAS first locates all the task-unit classes in the system. These classes are generally threads or workers, such as classes that extend `Thread` or `Runnable` in Java. The analyzer then runs ASV inference on each task-unit class.

Algorithm 1 lists the core algorithm. It starts by inferring the SVs in the code (Line 2). `InferCSV` simply treats all *non-static*, *non-constant* fields defined in a task unit to be SVs.

LEGOLAS then analyzes the main task method of the task unit class, such as the `run()` method of a `Thread`. It finds the basic blocks in the task method that are control dependent on some SV and treats each such basic block as a new ASV.

Specifically, the analyzer iterates through instructions in the task method. Upon a branch instruction, it checks if the branch condition is dependent on some SV (Line 13). This check considers not only direct usage of SV but also indirect data dependence, *i.e.*, a branch condition involving a local variable that gets its value from an SV. Accordingly, the analyzer builds a data dependence graph of the SVs (Line 4). The algorithm then recursively processes the basic blocks control dependent on this branch instruction (Line 15). A system should perform some non-trivial actions in an abstract state. Thus, the analyzer ignores basic blocks that only perform assignments or call basic functions like `toString` (Line 20).

Once the proper basic blocks are located, the analyzer assigns indexes for them, asv_0, . . . , asv_n. The indexes are local to the task-unit class.

Note that our ASV inferrence is *not* equivalent to conventional control-flow path. The ASVs instead collapse program paths into more meaningful ones to guide fault injection.

For each inferred ASV, the analyzer instruments a call to the LEGOLAS agent. At runtime, the agent notifies the LEGOLAS state tracker of the method and asv_i that are entered, along with the node id, the name and id of the current thread.

Example. Figure 8 shows the ASVs LEGOLAS infers and inserts for the code in Figure 5. The inferred ASV is slightly different from S_2 in the simplified snippet in Figure 7. This is because the `logCount` is a local variable, thus the LEGOLAS analyzer does not treat it as an SV. Another variable `snapThd` is a non-static field of `SynchRequestProcessor`. LEGOLAS treats it as an SV and infers the asv_3 that represents the snapshot stage. This result is in fact more accurate than using `logCount`, because it infers an additional state—asv_2—a previous snapshot is ongoing while the snapshot threshold is reached.

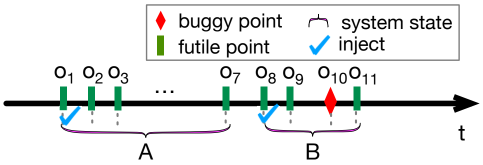


Figure 9. The buggy point may not be the first request in a state.

Alternative. We also explored other ASV inference methods. For example, we observe that although some function only uses local variables, it can still represent an important system service stage, *e.g.*, handling an event. Defining an ASV at the function entry can be useful. We chose our above inference method for its simplicity. As Section 7 later show, it is general enough to apply on all the popular distributed systems we evaluate and achieve significant performance.

5.4 Injection Decision Algorithm

With the inferred abstract states, LEGOLAS enables *stateful* decision policies for efficient fault injection. When the controller receives an injection request from the LEGOLAS agent, the controller checks which abstract state the target system is in at the time of the injection request to make a decision.

A straightforward stateful policy is to grant an injection request only if the system is in a new state. We call this a *new-state-only* policy. While this policy matches the intuition that complex bugs are often only triggered when the system enters an unusual condition, it has several drawbacks. As Figure 9 shows, there can be multiple injection requests from one state, and a buggy point may not be the first request. Indeed, for the ZooKeeper example, even though the bug only appears in the snapshot state, the snapshot function performs several write operations for the ACL before it reaches the buggy point. Injection at these points does not expose anything interesting. This policy also heavily relies on the abstract state analysis to be precise. If the static analysis misses instrumenting an ASV in the scope close to the buggy point, the buggy point will likely be treated as in a seen state. In addition, it may take a long time for the system to enter a new state. If we keep waiting for a new state, we may not inject anything when the workload finishes and waste an injection trial.

To address these drawbacks, we design a **budgeted-state-round-robin** (*bsrr*) policy. Algorithm 2 lists its algorithm.

The algorithm allocates a budget (default 5) for each state to be potentially granted injection more than once. This relaxes the stringent new state requirement. After *all* states use up their budgets, the budgets are reset (Line 2).

Algorithm 2: Budgeted state round robin (bsrr) policy

```
   Global Vars: Queue<State> rrl, Map<State,Info> visit
   /* invoked at start of a fault injection trial */
1  Function setupNewTrial():
2      resetIfAllUsed(rrl, visit);
3      while !rrl.empty() do
4          $s \leftarrow$ rrl.pop();
5          info $\leftarrow$ vist.get(s);
6          if info = nil or info.budget > 0 then
7              rrl.append(s);
8              break;
9          end
10     end
11     while !rrl.empty() and visit.get(rrl.front()).budget = 0 do
12         rrl.pop();
13     end
14     updateProbabilities(visit);
   /* invoked for each injection request */
15 Function shouldInject(request):
16     curr $\leftarrow$ getCurrentState(request);
17     if not visit.contains(curr) then
18         visit.put(curr, new Info());
19         rrl.append(curr);
20     end
21     info $\leftarrow$ visit.get(curr);
22     info.occur $\leftarrow$ info.occur + 1;
23     if rrl.front() $\neq$ curr then return false;
24     if info.budget > 0 and rand() < info.prob then
25         info.budget $\leftarrow$ info.budget - 1;
26         return true;
27     end
28     return false;
```

It keeps a round-robin list of the abstract state tuples (`rrl` in Algorithm 2). Suppose the list has $s_1, s_2, \ldots, s_n$. The algorithm intends to grant injection requests from state s_1 for the first trial, grant requests from s_2 for the second trial, and so on. In other words, it **focuses on** one state in one trial.

Specifically, before each trial, *bsrr* rotates the state focused in the last trial to the of the round-robin list (Line 7). If a state's budget is used up, it is removed from the list.

The round-robin design of *bsrr* addresses the imbalanced injections problem illustrated by Figure 10: injection requests from different tasks occur in parallel; in all three trials, operations in the frequent state A are injected (within its budget), while B and C are starved.

The algorithm also applies randomization to allow exploring different choices when a state has multiple injection requests. The probability p should be set properly. If it is too large, we would always grant the first (few) requests in a state. If it is too small, we may waste the injection trial.

We calculate p for each state tuple based on c—the times this state appears in injection requests. We set $p = 1 - e^{ln(0.01)/(c+1)}$. This formula's rationale is that we want to (i) grant at least one injection among the c requests to avoid wasting the trial; (ii) let the injection occur neither too early nor too late among the c requests. The probability that all c injections are *not* granted is $(1-p)^c$. Because of (i), this probability should be close to 0. Suppose $(1-p)^c = \epsilon$. With (ii), ϵ should not be too small; otherwise p is too close to 1 and the injection would be too early. We set ϵ to 0.01, and solve this equation, which gives $p = 1 - e^{ln(0.01)/c}$. We use $1 - e^{ln(0.01)/(c+1)}$ instead to handle corner cases of $c = 2$ or 3.

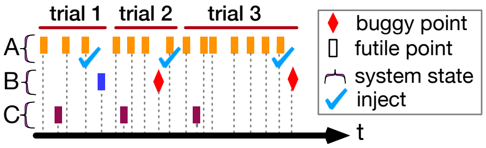


Figure 10. All injection chances are given to operations in state A.

The *bsrr* policy is adaptive to leverage information from prior trials. Upon each injection request, the algorithm dynamically updates the parameter c (Line 22). Before a trial starts, it uses the occurrences from previous trials to re-calculate the probabilities for the visited states (Line 14). Similarly, *bsrr* updates the round-robin list dynamically (initially empty). If a state in an injection request is not visited before, it is added to the round-robin list for later exploration (Line 19).

5.5 Dynamic Experiment

Legolas starts a fault injection experiment by first instantiating a Legolas server composed of a controller, state machine tracker, orchestrator, and failure checker (Figure 2). The Legolas server exposes RPC interfaces to the Legolas agent embedded in the target system.

The experiment proceeds in continuous trials. In each trial, the orchestrator starts a cluster of the instrumented system. After receiving registrations from all system nodes, the orchestrator invokes the workload driver, which creates clients that feed workloads to the target system. A client is terminated when it finishes its workload or times out or encounters exceptions. While the orchestrator restarts the target system in each trial, the Legolas server persists throughout the experiment and carries its recorded information across trials.

When a node enters a new abstract state, the Legolas agent notifies the state machine tracker. Each state update event is a tuple of `<node-id`, `sm-name`, `sm-instance`, `state-id>`. The `sm-name` and `-instance` are the task unit (usually a thread) class name and instance hash code. The Legolas tracker maintains the State Machines (SMs) for each node.

When a node reaches an injection hook, the Legolas agent sends an injection query to the controller, which is a tuple of `<node-id`, `sm-instance`, `operation-signature`, `fault-ids>`. Given an injection query, the controller obtains the associated abstract state by indexing the query's `<node-id`, `sm-instance>` to the SM map in the state tracker. Then the controller makes decisions based on the state and injection query.

In each trial, Legolas by default grants at most one injection. Allowing multiple injections in a trial only requires a simple configuration change. While it seems more attractive to keep injecting faults until the system fails, that choice has several disadvantages. First, although distributed systems are designed to be fault-tolerant, each system has a limited tolerance level. If we keep injecting in a single run, the system may likely break as expected. Second, each injected fault can alter the system state and leave side effects. With continuously injected faults, it becomes very difficult to judge the system behavior and tell which fault is responsible for the symptom. Third, if injections are performed in a non-stop run, we may go deeper in an execution path, but never get to backtrack and explore injecting earlier, skipped operations or other paths.

5.6 Workload Driver and Failure Checkers

To exercise a system, we write a workload driver by adapting from its representative test cases. The driver creates several clients that proceed in phases, *e.g.*, create keys, write and read. Each client typically interacts with one node and reports its observations (timeouts, errors, or successes) to LEGOLAS. We currently use a relatively small workload scale, such that a trial does not take long and LEGOLAS can explore more trials.

At the end of a trial, the LEGOLAS orchestrator summarizes the reported result from each client such as the actual progress of a workload phase. It further collects each node's status from the system logs and OS signals, *e.g.*, whether the node fails to start or crashes or exits with fatal exceptions. The system stack traces are also recorded when a fault is injected.

LEGOLAS also includes failure checkers to judge the injection result based on the information collected by the orchestrator. Checking crash failures is simple. Judging partial failures is more complex and difficult to be fully automated. Our checker provides first-level classification.

The checker first tags the injection trials by the workload progresses. Those successful trials and unambiguously expected failed trials (*e.g.*, shutdown due to faults in reading metadata) are classified as normal. Noisy trials (*e.g.*, some client requests fail despite no fault being injected) are filtered.

It then ranks the remaining trials based on the differential observability principle in state-of-the-art partial failure detection work [24]. In particular, it highlights trials in which (1) a fault is injected in one node, but *only* another node's clients report errors; (2) the system's own detector indicates a node is active, but the node's clients report errors; (3) only a subset of clients fail to complete their workloads. In addition, the checker clusters the trials by similarities in the collected stack traces so that similar symptoms are investigated together.

Users can then inspect the highly suspicious trials and read the relevant code to confirm bugs. Developers of a system can add system-specific checkers to more quickly identify bugs.

6 Implementation

We implement LEGOLAS with around 7,500 SLOC for the core components, and 100–300 SLOC for the workload driver for each evaluated target system. The LEGOLAS analyzer is built on top of the Soot [48] program analysis framework, so it supports systems in JVM bytecode, including Java and Scala. Its core analysis algorithms are based on universal programming language constructs such as thread classes, member variables, branches, and conditionals. Thus, they are language-independent. The controller and orchestrator are designed in a client-server architecture using Java RMI for local RPCs.

System	Release	SLOC	Type
ZooKeeper	3.6.2	95K	Coordination service
HDFS	3.2.2	689K	Distributed file system
Kafka	2.8.0	322K	Event streaming system
HBase	2.4.2	728K	Distributed database
Cassandra	3.11.10	210K	Distributed database

Table 1. Evaluated distributed systems in latest releases.

7 Evaluation

Our evaluation aims to answer several key questions: (1) does LEGOLAS work on large distributed systems? (2) can LEGOLAS expose new complex fault-triggered bugs? (3) does the abstract states LEGOLAS infers significantly help the fault injection efficacy? (4) how does LEGOLAS using the *bsrr* policy compare to other policies and state-of-the-art solutions?

Evaluated Systems. We evaluate LEGOLAS on the latest stable releases of five popular, large-scale distributed systems (Table 1). These systems have different functionalities, written with various programming paradigms. Our supplementary material lists the workloads we use in the testing.

Measure. For a testing tool, its ability to exposes new bugs is a key measure. Our evaluation thus centers around this aspect (Section 7.2). Since our target systems are widely deployed in production and have been extensively tested for years, finding new bugs in their latest releases is not an easy task.

As an additional measure, we also apply LEGOLAS on a number of known bugs in old releases of the systems (Section 7.6), including the running example bug in Figure 1.

Setup. We run experiments on servers with a 20-core 2.20GHz CPU and 64 GB memory running Ubuntu 18.04.

Each system's fault injection experiment consists of 2,000 trials. A trial's time is dominated by the target system startup and workload execution. The trials' durations vary depending on how the system reacts to the injected faults and whether it fails early or not. The experiment time for the five systems is 2.7 hrs, 10.7 hrs, 23.5 hrs, 8.4 hrs, and 54.6 hrs respectively.

We use the *bsrr* policy (Section 5.4), and set the state budget to the default value of 5 *for all systems*.

Due to the large scale of experiments and time constraints, our testing focuses on the following faults: (1) I/O related exceptions, e.g., `FileNotFoundException`, `SocketException`; (2) custom exception types that inherit from `IOException`; (3) delays to function calls that involve disk or network I/O. We run two separate experiments for each target system.

System	Class	SM	ASV				Static. Injected	
			Total	Mean	Min	Max	Methods	Points
ZK	643	37	230	6	1	31	236	632
HDFS	4636	104	390	4	1	16	1006	2328
Kafka	5829	51	220	4	1	15	302	682
HBase	10462	96	312	3	1	17	5280	10259
CSD	3630	125	314	3	1	18	552	1308

Table 2. Statistics of applying LEGOLAS static analyzer on the target systems. *Class*: analyzed Java classes; *SM*: identified state machine classes; *ASV*: abstract state variables created in each SM.

7.1 Injection Points and Abstract States

LEGOLAS successfully applies on the five systems. Besides scaffolding setup (e.g., class paths, task-unit class types), the analyzer does not require special input when applied on a new system. The injection policies are also not specially tuned.

Table 2 lists the statistics from applying the LEGOLAS analyzer on the target systems. We can see that the number of state machines (SM) LEGOLAS extracts is much smaller compared to the total number of classes in the system. In addition, the numbers vary across different systems due to their design choices. For example, ZooKeeper has a relatively small number of SMs, while Cassandra has over 100; yet, ZooKeeper has the largest average number of abstract state variables (ASV) in the SMs. This is because ZooKeeper is designed with a relatively small number of heavy threads, while Cassandra adopts an event-driven architecture that uses many short-lived runnables. The average number of ASVs per SM is moderate, because currently the LEGOLAS analyzer only analyzes the direct SM classes and the entry methods. Deeper analyses are possible, but even with the current analysis, the LEGOLAS injector already gains significant advantages.

7.2 Finding New Bugs

Our overall experience in the fault injection experiments is that the evaluated systems are robust to tolerate or at least cleanly abort various faults in most places. Take ZooKeeper as an example. If a thread is doing a socket write and a network delay is injected, this thread will get stuck. In general, ZooKeeper can handle the fault correctly even though this thread hangs. For example, if the `LearnerHandler` thread hangs in this way, the `QuorumPeer` is able to confirm the stale `PING` state and abandon the problematic `QuorumPeer`.

Nevertheless, LEGOLAS still finds new bugs in *all* tested systems. In total, it finds 16 unique new bugs (Table 3). All bugs are non-trivial and require domain knowledge to understand, such as mishandling of errors, design flaws, and synchronization issues. They all trigger partial failure symptoms.

We reported the bugs to developers. Four reports are marked as critical, ten reports as major, and two as normal. Ten reports have been explicitly confirmed by developers.

Our bug reports generate substantial discussions with developers, with a median of 21 comments and a maximum of 42 comments. Our reports to ZooKeeper inspired the developers to adopt fault injection testing practice.

Bug Id	Symptom
ZK-1	requests to one follower get stuck and the follower cannot rejoin the quorum for a long time
ZK-2	one follower keeps trying to join the quorum but keeps failing, even though the other 2 nodes get the request
ZK-3	one follower takes a long time to join the quorum and causes temporary service unavailability
ZK-4	causes re-election and partial service unavailability
KA-1	some client requests to create topics experience InvalidReplicationFactorException errors for a long time
KA-2	a broker proceeds but consumers hang for more than 3 minutes without any error log
KA-3	some clients get unexpected TopicExistsException even though they have never created the topic before
HDFS-1	some client hangs instead of timing out after ping interval
HDFS-2	normally a client is immediately notified of the error, but now the client hangs for 1min
HDFS-3	some client hangs forever without any log and the expected file does not exist in HDFS
HDFS-4	namenode hangs even with the async edit logging
HDFS-5	some client hangs for a long time
HB-1	table creation command hangs for a long time without any error but list command shows the table exists
HB-2	some table create requests experience a long delay
CS-1	clients to node 1 experience sporadic CQL operation timeout due to unconfigured table
CS-2	node continues after erroreous startup state and later causes client failures

Table 3. New bugs found by LEGOLAS (real bug ids are obscured for review anonymity rules). All issues cause partial failure symptoms. The root causes are diverse. Our supplementary material lists the root cause and injected fault for each issue.

Case Studies. *ZK-1* In one trial, LEGOLAS injects a delay to a network write operation inside the `Learner writePacket` method, which causes the entire follower to be stuck. Unlike the other situations in which the fault could be tolerated by ZooKeeper, `QuorumPeer` itself and other threads did not detect or handle this issue. The root cause is that although the `readPacket` method sets a socket timeout, the follower still did not initiate the rejoin because the `QuorumPeer` is blocked in `writePacket` and would not proceed to the receiving stage, so no timeout exception is thrown to trigger the error handling. The state for the injection is sending the request to the followers from `FollowerRequestProcessor`. LEGOLAS infers the state machine of `FollowerRequestProcessor` and obtains the ASVs. One ASV exactly points out this state. The other ASVs are also meaningful and represent different stages of the request processing in `FollowerRequestProcessor`, such as forwarding the request to the next cascading processor and exception handling.

HDFS-2 In one trial, LEGOLAS injects an `IOException` in the `BlockReceiver` module while one datanode is forwarding the data blocks to a mirror (another datanode). The clients

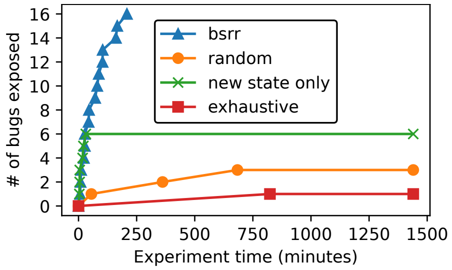


Figure 11. Efficiency of decision policies in Legolas on detecting new bugs. *bsrr*: our budgeted-state-round-robin algorithm.

get stuck without any error log. After some investigation, we found that normally the datanode in such a situation will inform the client of this error state immediately, and then the client will resend the blocks. This process would be fast. Through careful code analysis, we find the root cause for the symptom Legolas exposes is a complex timing bug. In particular, when the datanode encounters the `IOException` it sets the `mirrorError` flag. However, a concurrency condition exists in which the `mirrorError` flag set could be shortly after the `PacketResponder` checks this flag, causing `PacketResponder` to not notice this error status and get blocked, and the `ACK` packet will not be sent by the mirror datanode.

HDFS-4 In one trial, Legolas injects an `IOException` in `FSEditLogAsync` thread. The injection occurs when it sends the responses to the client and other servers, after it commits the transactions. The injected exception is not properly handled and without retry. Later on, a client hangs because it is a critical operation in the RPC and has no chance to be compensated now. Legolas infers the state machine of `FSEditLogAsync` and obtains the ASVs. One ASV exactly represents the state of sending responses to client, even if the main service of `FSEditLogAsync` is commiting the transactions to the disk.

7.3 Impact of Abstract States and BSRR

This paper's key thesis is that our inferred abstract states can enable efficient fault injection. Section 7.2 shows that Legolas finds complex new bugs with our *bsrr* algorithm. To further validate our thesis, we compare the *bsrr* algorithm with alternative decision policies on the 16 new bugs.

For each policy, we run a 2000-trial experiment and measure the number of bugs it exposes, as well as the time it takes to expose the bugs. The latter is an important metric. If a solution cannot expose a bug within a reasonable time, developers in practice likely will not use it even if the solution in theory may expose the bug after a long time.

Figure 11 shows the result. The exhaustive policy only exposes one bug in the 2000 trials. The random policy only exposes three bugs. It is also very inefficient. The median and maximum time for finding the three bugs are 362 minutes and 679 minutes, respectively. After finding the third bug, it fails to find more bugs after the experiment continues for 24 hours.

	Detected Bugs	Median Detection Time
FATE [18]	1	1057.9 minutes
CrashTuner [38]	4	20.4 minutes

Table 4. Effectiveness of existing work on the 16 new bugs.

The new-state-only policy exposes six bugs, with a median time of 11.4 minutes. It is the best among the baseline policies, showing the advantages of the abstract states.

The *bsrr* policy significantly outperforms all alternatives. It exposes 16 bugs in as quickly as 6.8 minutes, and a median time of 58.2 minutes (maximum 152.5 minutes). It is robust to expose much more bugs while achieving good efficiency.

7.4 Comparing with Other Solutions

Research Baselines We compare Legolas with two state-of-the-art fault injection research projects, FATE [18] and CrashTuner [38]. FATE tests *multiple failures* by using a concept of *failure IDs* to efficiently enumerate the combinations of failures. CrashTuner uses *meta-info variable accesses* to decide the timing of injecting node crashes for exposing crash recovery bugs. Both works focus on coarse-grained node-level faults, making them not directly comparable to Legolas.

We apply their key ideas to attempt meaningful comparisons. In particular, we define the *failure IDs* as described in the FATE paper and implement a policy in Legolas to grant an injection request when its associated failure ID has not been visited. For CrashTuner, because its analyzer component is not available, we re-implement its static analysis to identify all meta-info variable accesses and assign each access point a global ID. We instrument each access point to record the accesses at runtime. Then we grant an injection request when some meta-info variable access occurs within the past 5 ms and the access ID has not been seen. The latter is needed because a system may access the meta-info variable in a deterministic order, leading to only one injection being always granted in an experiment if the access ID is not checked.

Table 4 shows the result. FATE's failure ID scheme only detects one of the 16 new bugs in 1057.9 minutes. CrashTuner's meta-info variable scheme only detects four bugs. Legolas significantly outperforms both solutions.

Popular Tool Baselines We further compare Legolas with three fault injection tools that are popular among developers: `CharybdeFS` [2] (a fault-injection filesystem), `tcconfig` [4] (a network fault injection tool based on Linux Traffic Control), and `byte-monkey` [1]. Byte-monkey is closer to Legolas in that it also performs bytecode-level fault injection.

These tools rely on user-provided parameters to configure the injection, such as the packet loss rate and probability of returning an error code. Settings that are too large or too small produce meaningless results. We choose one moderate setting and one mild setting for each tool. We exercise the target systems using the same workloads from Legolas and run each tool with a 2000-trial experiment.

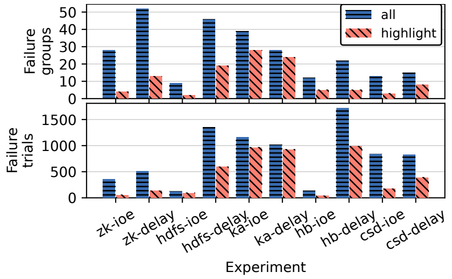


Figure 12. Number of groups that the Legolas failure checkers classify the trials with errors into.

Most injections lead to either a high percentage of successful trials or a high percentage of early exits (shown in supplementary material). For the small percentage of *partial_progress* injection trials (mostly from `tcconfig`), the failed client requests either happen directly because of the injected fault (*e.g.*, the server logs that it is unable to read data from client) or the system is in the middle of fault handling and successfully recovers. We verify that none of these trials expose any of the 16 new bugs Legolas finds. We also vary the parameters, but the conclusions remain the same.

In the Legolas decision policy comparison experiments (§7.3), its random policy exposes three bugs. In comparison, the evaluated popular tools do not expose any bugs with their random strategies. The discrepancies are due to the probability factor and the fact that Legolas's in-situ injection mechanism has more precise control—it instruments operations that are possible to throw `IOException` (or its subtype) errors, which may be caused by complex environment errors. For instance, while `tcconfig` injects packet loss, it fails to trigger errors for function calls that internally throw `IOException` only upon a sequence of network errors. Also, its injection has a low chance of affecting only a special subset of operations.

7.5 Manual Effort and False Positive

To use Legolas on a new system, the analysis and instrumentation are fully automated. The fault injection experiment also does not require manual tuning. Workload drivers are based on existing test cases and re-usable across versions.

The main manual effort is to confirm a bug after testing. However, this is not a unique requirement of Legolas, but rather common among testing tools.

For each failure trial, Legolas records detailed stack traces of the injection and failure point along with the client progress and server logs. Our failure checker (Section 5.6) highlights trials with suspicious partial failures. It also group failure trials that have similar behavior. Figure 12 shows the classification result for each experiment. There are tens of failure groups, with a small portion of them being highlighted. Typically only one or two trials in a group need to be examined.

Those designs make the overall inspection efforts moderate. It roughly takes a graduate student author one to two weeks per system to examine the failure groups, read and understand the relevant source code, and confirm where the bug is.

	ZooKeeper	HDFS	Kafka	Cassandra	HBase
w/ i.i.a	0	0	0	0	0
w/o i.i.a	45 (6)	20 (9)	0	894 (10)	86 (10)

Table 5. Number of trials that have invalid injections, with and without the invalid injection analysis in Section 4.1. The numbers in parentheses are the unique locations of these invalid injections.

System	Bug Id (Exposure Time)
ZooKeeper	ZK-2029 (15.4 min), ZK-2201 (30.6 min), ZK-2247 (52.1 min), ZK-2325 (2.6 min), ZK-2982 (18.5 min)
Cassandra	CA-6364 (10.0 min), CA-6415 (330.6 min), CA-8485 (25.3 min), CA-13833 (86.7 min)
HDFS	HDFS-11608 (29.2 min), HDFS-12157 (39.9 min)

Table 6. Legolas exposes *known*, real bugs in old releases.

Developers of these systems can inspect much faster. When using Legolas themselves, they can also add system-specific failure checkers to speed up the bug confirmation.

For a fault injection tool, a false positive happens if the fault being injected turns out to be impossible in reality. We observe such false positives and they are caused by the same problem: Legolas injects an `IOException` to a function that declares `IOException` in its signature but cannot possibly have I/O errors. For example, we inject an `IOException` to the `writeBoolean` call inside the Cassandra `serialize` method. This injection triggers a buggy symptom. However, this method is using a memory buffer for the `writeBoolean` call.

As described in Section 4.1, the analyzer in Legolas performs an inter-procedural, context-sensitive analysis to identify such false injections. Table 5 shows the number of false injection trials in a 2000-trial experiment for each system with and without this analysis. The result shows that the analysis successfully eliminates all of these invalid injections.

Another potential source of false injection is the automatic exception instance creation (Section 4.2). Our method may create some invalid exception instances. However, we did not observe such false positives in our experiments.

Note the failure trials in our experiments only mean that the target system and/or clients experience some issues with the injected faults. For example, the tool injects a fault in node 1, and the clients connected to node 2 experience write timeouts. This behavior may be expected. Such a trial does not mean the fault injection causes a false positive. The injected fault is legitimate. The system's reaction to the fault is by design. The injected fault just does not expose a bug. Therefore, they are ineffective bug-finding attempts rather than false positives.

7.6 Reproducing Known Bugs

Besides finding new bugs, we further evaluate Legolas's capabilities on reproducing known partial failure bugs. We collect 11 *real-world* partial-failure issues from older versions of three evaluated systems. As Table 6 shows, Legolas can relatively quickly reproduce these known bugs.

	ZooKeeper	HDFS	Kafka	Cassandra	HBase
w/o i.i.a	8.5 s	29.4 s	35.9 s	19.9 s	72.8 s
w/ i.i.a	8.9 s	31.6 s	36.9 s	20.9 s	77.6 s

Table 7. Time of static analysis and instrumentation, without and with the invalid injection analysis (§4.1).

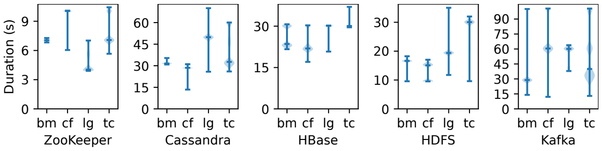


Figure 13. Distribution of one fault injection trial duration. *cf*: CharybdeFS; *bm*: byte-monkey; *lg*; Legolas; *tc*: tcconfig.

7.7 Performance

We measure the fault injection trial duration for different tools. Legolas is expected to be slower than the baselines because its fault injection queries are dispatched through RPC requests to the injection controller. It also tracks the abstract states.

Figure 13 shows the distribution of trial durations for the four tools. In Cassandra and HDFS, the Legolas's trials show larger duration ranges, but even the maximum (70 seconds) is still acceptable; in ZooKeeper and HBase, the trial durations of Legolas do not have significant differences from the other tools. One reason is, as mentioned earlier, a trial duration's major bottleneck is in the server node creation and workload execution, and it depends greatly on how the system reacts to faults. In addition, Legolas uses local RPCs based on Java RMIs. Our microbenchmark shows the RMI latency is between 10 μs–50 μs. The *bsrr* policy function for one injection request has a median latency of 3 ms.

Table 7 shows the performance of the Legolas static analysis and instrumentation. The longest time (73 s) is in analyzing HBase, which has the largest codebase size. The fast analysis and instrumentation make Legolas suitable to integrate in regular fault injection testing. With the invalid injection analysis enabled (§4.1), the total time increases slightly.

8 Discussion and Limitations

Our ASV analysis can be enhanced by analyzing not only states in thread classes but also states that are passed to other classes through function calls. Also, the controller currently only consider SMs individually. Considering the combination of different SMs can potentially improve injection decisions.

Our workload drivers use workloads on a relatively small scale to exercise the target system. More workloads can be added to the Legolas workload drivers, which is not a difficult task and would allow Legolas to expose fault-induced bugs that require large workloads (e.g., performance bugs).

Legolas injects a single fault in one fault injection trial. It would miss partial failure bugs that are triggered by multiple faults. Supporting injection of multiple faults in Legolas only requires small changes. However, the decision algorithm and failure checkers would likely require significant changes.

Legolas does not explicitly control non-determinism in the target system, such as thread schedules, which is the focus of concurrency testing tools. Thus, while Legolas can expose a concurrency bug, it may not expose it reliably or efficiently. Legolas can be combined with concurrency testing tools.

9 Related Work

Fault Injection Early work on fault injection targets standalone software [22]. Faults such as CPU bitflip and disk error are injected into the hardware, simulated environment, or libraries (LFI [41]). Fault injection testing becomes a popular practice in distributed systems with much research [3, 9–11, 13, 16, 18, 26, 27, 38, 42, 43]. These solutions typically inject coarse-grained faults externally, and are designed to catch a specific type of bugs such as crash recovery bugs. They perform injections randomly, or exhaustively, or rely on manual specifications. Some [13, 38] inject faults in special code regions that are assumed to be likely buggy, such as when accessing meta-info variables.

Legolas injects fine-grained faults *in-situ* in a system. It designs a novel method to automatically infer abstract states from distributed system code. Its new budgeted-state algorithm efficiently explores the large injection space. Legolas can expose a variety of complex partial failure bugs.

TSVD [32] is a delay-injection tool designed to detect thread-safety violations in multi-threaded programs. It tracks the near-misses and injects delays only at dangerous locations that may cause thread-safety violations. Sieve [47] is a tool designed for testing cluster-management controller, a special type of distributed software. It injects crashes and delays to perturb a controller's view of the cluster state and finds bugs by comparing the cluster state. Legolas is an orthogonal effort compared to TSVD and Sieve. It targets broader distributed systems and focuses on fine-grained faults using a new scheme based on abstract system states.

Model Checking Model checking enumerates the possible interleaving of non-deterministic events such as messages. It has been applied to distributed systems [21, 29, 31, 46, 49]. Some distributed system model checkers (dmcks) including MODIST [49], SAMC [31], and FlyMC [40] also explore the interleaving of crash/reboot failure events. Legolas shares high-level similarity with these solutions in that it systematically explores the fault injection space. However, Legolas is a complementary effort. Existing dmcks target protocol bugs caused by complex interleaving of node-level events, while Legolas targets non-protocol bugs triggered by diverse faults in fine-grained program instructions. Legolas can leverage a dmck to drive the target system into unexplored states, allowing Legolas to try more injections.

Distributed Concurrency Bug Detection Several projects [33, 35, 37, 50] aim to detect concurrency bugs in distributed systems. FCatch [35] applies happens-before analysis on correct execution traces to identify unprotected conflicting operations. LEGOLAS is a general fault injection framework aiming to expose diverse bugs. It is complementary to these solutions.

Partial Failure Detection Failure detectors are part of a running production distributed system to determine whether the system is faulty or not. Recent work [24, 36, 39, 45] explore advanced detectors for the notorious partial failures. LEGOLAS is an offline testing tool. It can leverage these advanced techniques in its checkers to find more bugs in testing.

10 Conclusion

This paper presents LEGOLAS, a fault injection testing framework that aims to catch complex partial failure bugs in large distributed systems. LEGOLAS uses static analysis to enable fine-grained, system-specific fault injection. It designs a novel method to extract abstract states from system code and uses them to efficiently explore the fault injection space. We apply LEGOLAS on five distributed systems and find 16 new bugs.

References

[1] Bytecode-level fault injection for the JVM. https://github.com/mrwilson/byte-monkey.

[2] CharybdeFS: A fuse based fault injection filesystem. https://github.com/scylladb/charybdefs.

[3] Jepsen: a framework for distributed systems verification, with fault injection. https://github.com/jepsen-io/jepsen.

[4] tcconfig: A tc command wrapper. https://github.com/thombashi/tcconfig.

[5] Kafka production failure because of BadVersionException. https://issues.apache.org/jira/browse/KAFKA-1407, 2014.

[6] updateisr should stop after failed several times due to zkVersion issue. https://issues.apache.org/jira/browse/KAFKA-3042, 2015.

[7] Kafka partial cluster breakdown. https://issues.apache.org/jira/browse/KAFKA-3577, 2016.

[8] M. Alfatafta, B. Alkhatib, A. Alquraan, and S. Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 351–368. USENIX Association, Nov. 2020.

[9] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 51–68, Carlsbad, CA, USA, 2018.

[10] P. Alvaro, J. Rosen, and J. M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 331–346, Melbourne, Victoria, Australia, 2015.

[11] C. Bennett and A. Tseitlin. Chaos monkey released into the wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html, 2009.

[12] H. Chen, W. Dou, Y. Jiang, and F. Qin. Understanding exception-related bugs in large-scale cloud systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 339âĂŞ351, San Diego, California, 2020.

[13] H. Chen, W. Dou, D. Wang, and F. Qin. CoFI: Consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 536âĂŞ547, Virtual Event, Australia, 2021.

[14] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 14:1–14:14, Santa Clara, California, 2013.

[15] M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A. Mritunjai, S. Perianayagam, T. Rath, S. Sivasubramanian, J. C. S. III, S. Sosothikul, D. Terry, and A. Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *Proceedings of the 2022 USENIX Annual Technical Conference*, USENIX ATC '22, pages 1037–1048. USENIX Association, July 2022.

[16] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST '17, page 149âĂŞ165, Santa clara, CA, USA, 2017.

[17] S. Ghosh, M. Shetty, C. Bansal, and S. Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 126âĂŞ141, San Francisco, California, 2022.

[18] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 238–252, Boston, MA, 2011.

[19] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, pages 1–16, Santa Clara, CA, USA, Oct. 2016.

[20] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 1–14, Oakland, CA, USA, 2018.

[21] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, Boston, Massachusetts, October 2011.

[22] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, Apr. 1997.

[23] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 73–90. USENIX Association, July 2022.

[24] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018.

[25] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI. ACM, May 2017.

[26] P. Joshi, H. S. Gunawi, and K. Sen. PREFAIL: A programmable tool for multiple-failure injection. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 171–188, Portland, Oregon, USA, 2011.

[27] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva. On fault resilience of OpenStack. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13, pages 2:1–2:16, Santa Clara, California, 2013.

[28] P. G. Kannan, N. Budhdev, R. Joshi, and M. C. Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, pages 253–268. USENIX Association, Apr. 2021.

[29] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 07)*. USENIX Association, Apr. 2007.

[30] K. Kingsbury and P. Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14(3):268âĂŞ280, Nov. 2020.

[31] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, page 399âĂŞ414, Broomfield, CO, 2014.

[32] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 162âĂŞ180, Huntsville, Ontario, Canada, 2019.

[33] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 677–691, Xi'an, China, April 2017.

[34] H. Liu, S. Lu, M. Musuvathi, and S. Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 155âĂŞ162, Bertinoro, Italy, 2019.

[35] H. Liu, X. Wang, G. Li, S. Lu, F. Ye, and C. Tian. FCatch: Automatically detecting time-of-fault bugs in cloud systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 419–431, Williamsburg, VA, USA, 2018.

[36] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574. USENIX Association, Feb. 2020.

[37] J. Lu, F. Li, L. Li, and X. Feng. CloudRaid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, FSE '18, pages 3–14, Lake Buena Vista, FL, USA, November 2018.

[38] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, and L. You. CrashTuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 114âĂŞ130, Huntsville, Ontario, Canada, 2019.

[39] R. Lu, E. Xu, Y. Zhang, F. Zhu, Z. Zhu, M. Wang, Z. Zhu, G. Xue, J. Shu, M. Li, and J. Wu. PERSEUS: A fail-slow detection framework for cloud storage systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST '23, Santa Clara, CA, USA, 2023.

[40] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa, A. Gupta, S. Lu, and H. S. Gunawi. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, Dresden, Germany, 2019.

[41] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, DSN '09, pages 379–388, June 2009.

[42] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller, and R. Padhye. Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 388âĂŞ402, Seattle, WA, USA, 2021.

[43] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 33âĂŞ50, Carlsbad, CA, USA, 2018.

[44] D. Nadolny. Debugging distributed systems. In *SREcon 2016*, April 7-8 2016.

[45] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, and H. S. Gunawi. IASO: A fail-slow detection and mitigation framework for distributed storage services. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 47âĂŞ61, Renton, WA, USA, 2019.

[46] J. Simsa, R. Bryant, and G. Gibson. dbug: Systematic evaluation of distributed systems. In *5th International Workshop on Systems Software Verification (SSV 10)*. USENIX Association, Oct. 2010.

[47] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu. Automatic reliability testing for cluster management controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 143–159. USENIX Association, July 2022.

[48] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[49] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, page 213âĂŞ228, Boston, Massachusetts, 2009.

[50] X. Yuan and J. Yang. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1141–1156, Lausanne, Switzerland, March 2020.